

KAPITEL 4

**OBJEKTORIENTIERTE
PROGRAMMIERUNG**

4 OOP - objektorientierte Programmierung

OOP ist vor allem bei großen Programmen sehr hilfreich, da ein Objekt sehr leicht eingebunden oder verändert werden kann. Dies verbilligt vor allem große Softwareprojekte enorm, da der Wartungsaufwand erheblich reduziert werden kann.

Was ist ein Objekt überhaupt? Ein Objekt hat Eigenschaften, die dieses Objekt einzigartig machen.

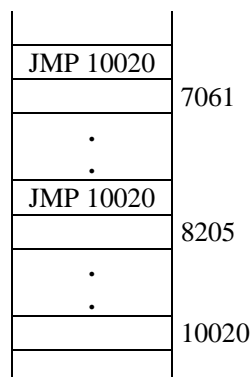
D.h. Ein Computer beispielsweise wäre ein Objekt, er hat einen Prozessortyp, einen Bustyp, eine Gehäuseart,... alle diese Attribute machen ihn (das Objekt) einzigartig.

Es stellt sich die Frage, wie die Daten zu speichern sind. Die beste Lösung ist eine Struktur, in der alle Daten, das Objekt betreffend, gespeichert werden. (1.Schritt in Richtung OOP).

4.1 Statisches Binden (static binding)

Welches Unterprogramm wo aufgerufen wird, entscheidet der Compiler.

```
if (...) roiser(...)
    → jmp 10020
```



4.2 Calling Overhead

Unterprogrammaufruf benötigt Zeit (bei wenig Zeilen ist ein UP unnötig)

4.3 Inline Definition

4.3.1 Macro Expansion

Code wird an die Stelle, wo er benötigt wird, kopiert → kein Stack, kein JMP, Zeitgewinn, verbraucht mehr Speicherplatz

4.4 Strukturen (structures)

Eine structure (in Pascal: records) entspricht einer Tabellendefinition. Das heißt, um die Übersichtlichkeit zu bewahren, dass zusammengehörige Variable ("Attribute") zu einer Struktur zusammengefasst werden. Dadurch spart man sich unnötige Variablendeklarationen.

```
typedef struct schueler {
    int alter;
    int edvnote;
    char name [..];
    float groesse;
};
```

```
h_alter ...
h_groesse ...
m_alter ...
m_groesse ...
x_name=h_name;
.
.
.
```

```
int a, b;
schueler huber, mayer, x;
```

```
a=7;
huber.alter=17;
huber.edvnote=3;
huber.name="Huber"
```

```
x=huber;
```

```
boolean ist_neg (schueler s)
{
    if (s.edvnote=5) return (TRUE);
    else return (FALSE);
}
```

```
schueler der_groessere (schueler a, schueler b)
{
    if (a.groesse>b.groesse) return (a);
    else return (b);
}
```

4.5 Vererbung (inheritance)

= eine Kurzschrift, um dem Programmierer die Schreibarbeit abzunehmen. Um ähnliche Objekte nicht extra definieren zu müssen, können Attribute und Methoden, zu den vorhandenen, mitvererbt werden.

```
struct kaschueler {
    alter
    groesse
    .
    .
    stenonote
};    schueler ⊕
int stenonote;
```

4.5.1 Mehrfachvererbung (multiple inheritance)

→ eine structure enthält mehrere structures (enthält nicht jede moderne Programmiersprache)

4.5.2 Selektivvererbung (selective inheritance)

→ Übernahme von gewissen Variablen (in keiner wichtigen Sprache vorhanden)

4.6 Generizität (genericity)

= änderbarer Datentyp in einer structure (für flexible Gestaltung; z.B. bei ADA)

```

struct usaschueler {
    schueler (char, int)
};
    ←
struct schueler (T, R) {
    alter (R);
    ...
    edvnote (T);
};
    →
struct kanschueler {
    schueler (float, float)
};
    
```

Hier handelt es sich für den Programmierer auch nur um eine Kurzschrift (beim Maschinencode ändert sich nichts).

4.7 Klasse (class)

= structure mit Unterprogramm(e)

```

struct schueler {
    alter ...
    .
    .
    boolean ist_neg ();
}
    {
    schueler der_groessere;
    }
    {
    if this.edvnote=5 ...
    }
    
```

boolean ist_neg (schueler s)
 {
 if s. ...
 ...
 }

bekommt einen Schüler

schueler y;

if y.ist_neg ⇔ if ist_neg(y)

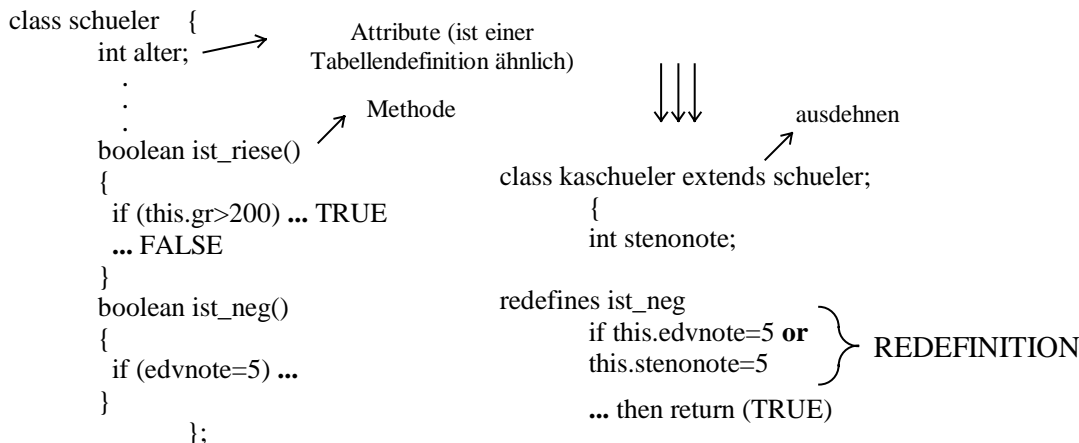
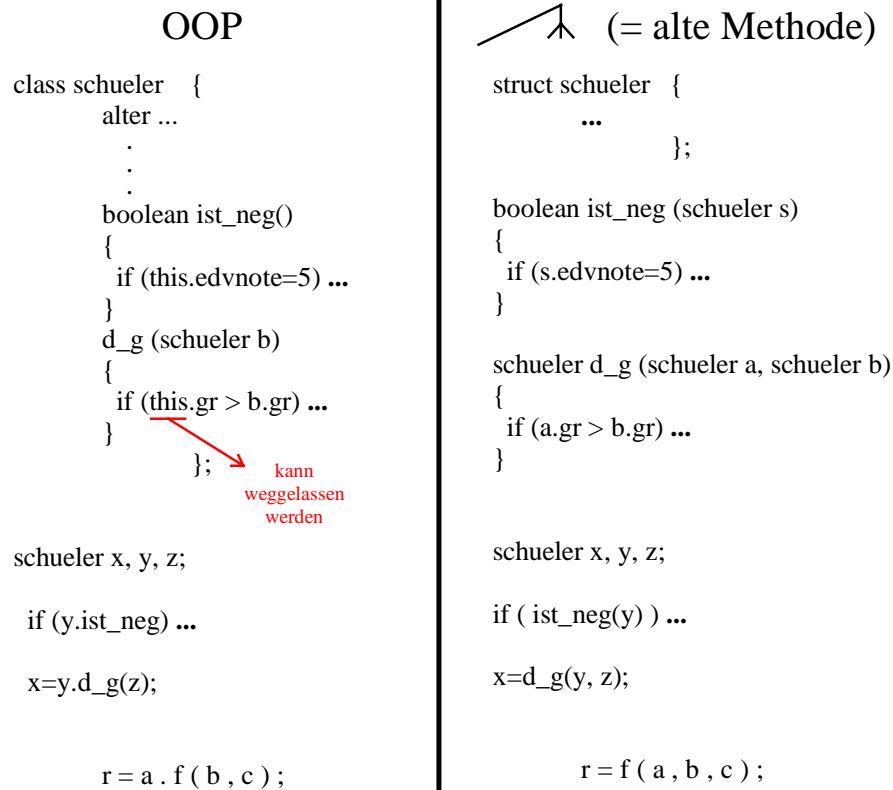
Der Vorteil der linken Variante ist der, dass bei einer Änderung von einem Unterprogramm zu einer Variablen (statt: ist_neg() {...} → int ist_neg;) keine Komplikationen auftreten.

4.8 Methode (method)

= ein Unterprogramm in einer structure (Smalltalk: MESSAGE)

Ich rufe mit dieser Struktur das Unterprogramm auf. → Ich rufe mit dieser Klasse die Methode auf.

4.8.1 Gegenüberstellung von OOP und alter Methode



```

class schueler {
    name : string;
    groesse : float;
    pnote : integer;
    ist_neg : boolean;
    {
        ... this.pnote=5 ...
    }
    ist_riese : boolean;
    {
        if this.groesse > 2.2
        then return (TRUE)
        else return (FALSE)
    }
};

schueler stz;

stz.name = "Stanzl";
stz.groesse = 1.75;
stz.pnote = 2;
if ( stz.ist_neg = TRUE)
    MsgBox ( ... )

```

```

schueler s, s2;
kaschueler k;

```

```

s := k;

```

akzeptiert (k hat mehr als s) links steht der Opa/Vater/Bruder;
man darf Kindern nichts zuweisen

```

k := s;

```

wird vom Compiler wahrscheinlich nicht akzeptiert,
weil *schueler* keine stenonote hat

```

if ( k.ist_stenogenie ) ...

```

STATIC TYPE = Typ, mit dem die Variable deklariert wurde
 DYNAMIC TYPE = Typ, den die Variable tatsächlich hat

Diese Compiler nennt man:

STRONGLY TYPED (streng getypt)

1. keine Zuweisung an Kinder
2. ob die Methode zulässig ist, entscheidet das *static type*

WEAKLY TYPED (Smalltalk)

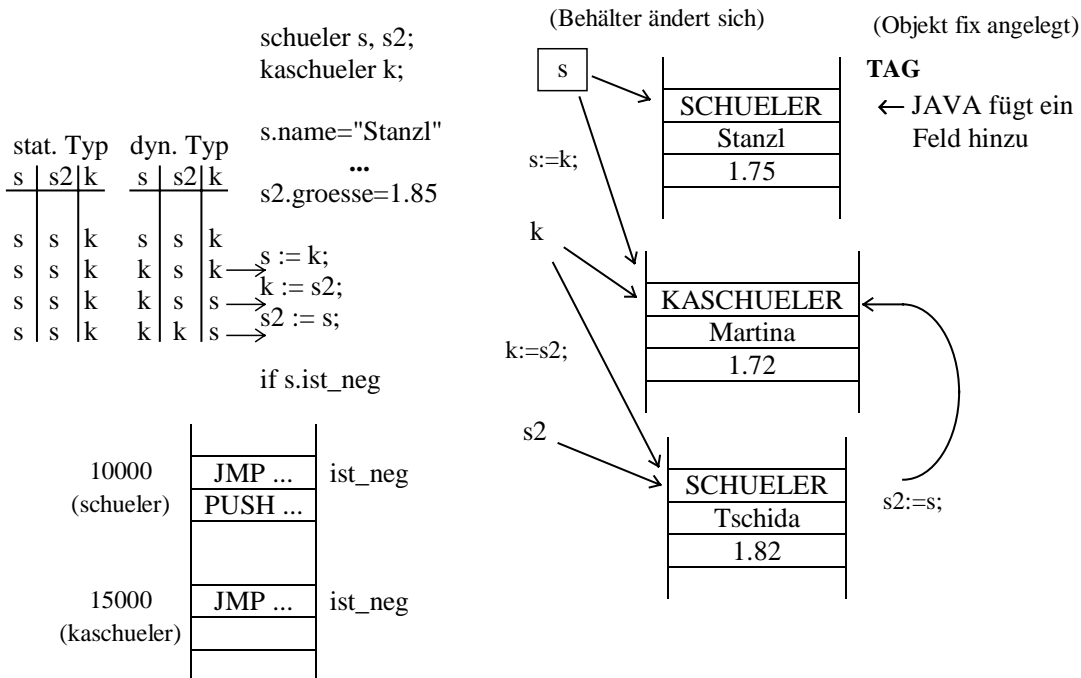
`s := k` hat den statischen Typ "schueler", aber nach der Zuweisung auch den dynamischen Typ "kaschueler". Von einem *strongly typed* Compiler werden aber nur die Zuweisungen (`s:=k`) akzeptiert, nicht die Ausführung der (nur bei dynamischen Typ) vorhandenen Methoden.

```

if ( InputBox () = 'YES' )   s:=k;
else                         s:=s2;
    .
    .
    .
if ( s.ist_stenogenie ) ...

```

Aufgrund der Benutzeraktivität ist der Compiler nie in der Lage, den dynamischen Typ im vorhinein zu bestimmen → Verlaß auf den Dispatcher

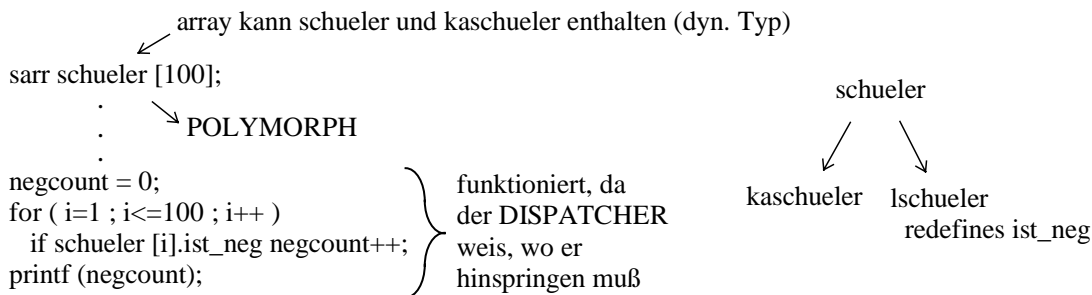


4.9 Dynamisches Binden (dynamic binding)

Der Compiler kommt zu einem Unterprogrammaufruf und weiß nicht ob er es ausführen soll und läßt während des Programmlaufes die Wahl einem kleinen Programm (dem DISPATCHER), das sich den TAG der Variablen anschaut und das jeweilige Unterprogramm auswählt.

→ auch late binding (z.B. bei C++, nicht bei Pascal)

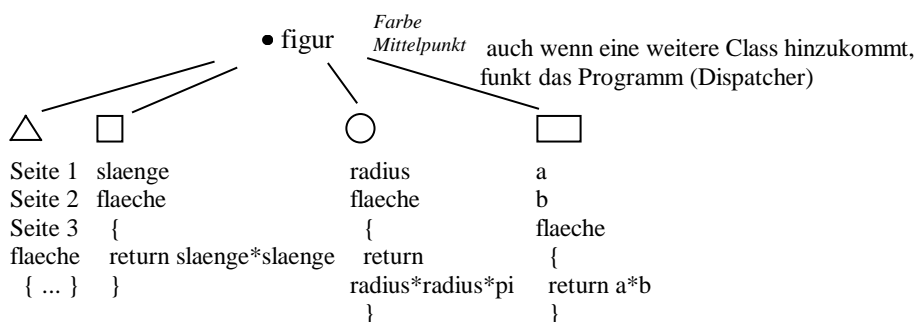
- EARLY BINDING
- LATE BINDING
- DYNAMIC BINDING



4.10 Polymorphie (Polymorphe Arrays)

Da bei Objekte Pointer benutzt werden, können verschiedenste Objekte in einem Array gespeichert werden. Auf den ersten Blick sieht es so aus, als würden verschiedenste Objekte in einem Feld gespeichert werden können; in Wirklichkeit wird jedoch jeweils nur ein Pointer auf ein Objekt gespeichert und dadurch wird die Polymorphie (Vielschichtig-/gesichtigkeit) ermöglicht.

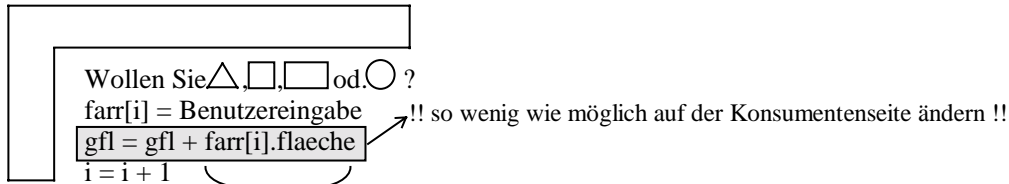
PRODUZENTENSEITE



KONSUMENTENSEITE

farr figur [100];

i=0; gfl=0

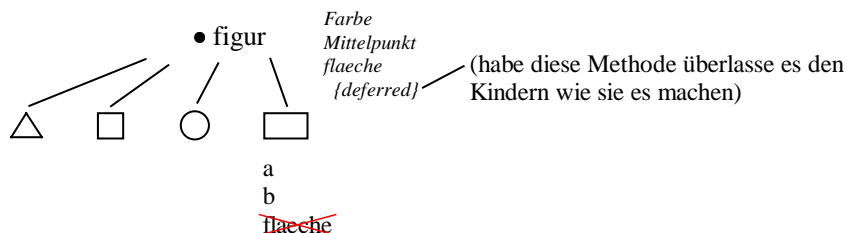


JMP ... (Compiler müßte fix entscheiden
z.B. JMP 1000 (akzeptiert nicht)
steht erst zur Laufzeit fest)

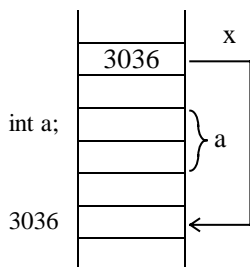
4.11 Aufgeschobene Klasse (deferred class)

- aufgeschobene Klassen sind ein Analyse-Tool
- OOP ist sein eigenes Analyse-Tool

Wenn die Kind-Klasse *Rechteck* keine *flaeche*-Methode enthält, hat der Dispatcher ein erhebliches Problem. Deshalb ist es besser, in der Vater-Klasse die *flaeche*-Methode zu definieren und dort DEFERRED (= aufgeschoben) anzugeben.



4.15 Konstruktor/Destruktor



ASCII 1 Byte
 UNICODE 2 Byte
 $2^{16} = 65536$

4.15.1 Dynamische Speicherverwaltung

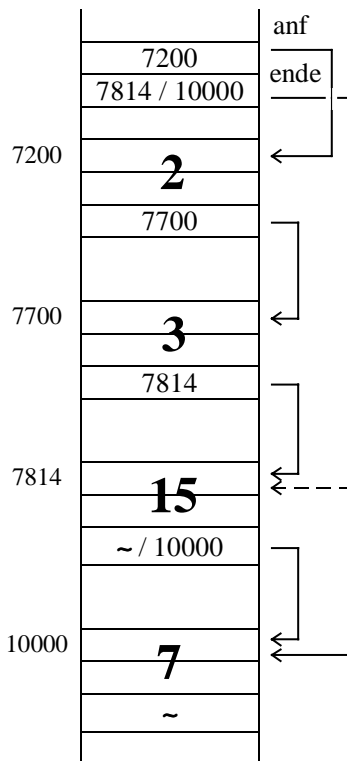
= Platzreservierung, wenn das Programm läuft
 in C: malloc (memory allocation / Speicher-Zuteilung)

`x = malloc (2)` während der Laufzeit werden 2 Byte an Speicher reserviert
`*x = 50 ;`

```
struct knoten {
    int zahl;
    knoten *next;
};
```

```
int *p;
int z=1;
```

```
while ( ... )
{
    z=InputDialog(...)
    if z<>0
    {
        p=malloc(sizeof(knoten))
        (*p).zahl=z
        (*ende).next=p
        ende=p
    }
    .
    .
    free p
    → Speicherrückgabe ans Betriebssystem
```



```
class knoten {
    int z;
    knoten *next;
    constructor knoten → wird bei new ausgeführt
    {
        this.next=NULL
        this.zahl=-1
    }
}
```

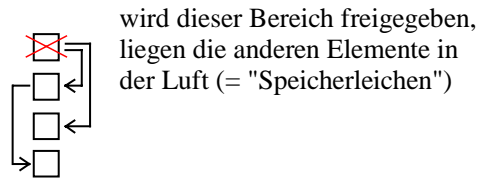
```
while ( ... )
{
    .
    .
    .
    p=new knoten
    p.zahl=z
    ...
    delete p
```

Kurzsymbol für **destructor** knoten → ~ knoten

Wenn eine Methode genauso wie die Klasse heißt, so ist sie ein Konstruktor
 → *constructor* kann weggelassen werden (C++, JAVA)

recursive dispose problem:
 = das Problem der Speicherleichen

garbage collection:
 = Wegräumen von Speicherleichen
 (JAVA)



best fit (beste):

sucht nach dem optimalsten Speicherplatz, es können jedoch kleine Speicherreste übrigbleiben, die unbenutzt bleiben (z.B. bei 30 Byte wird ein 32 Byte-Block benutzt)

worst fit (schlechteste):

Zugriff auf den größten Speicherblock, nimmt dadurch anderen Programmen, die den Speicher benötigen, den Platz weg (z.B. bei 30 Byte werden 1000 Byte angeschnitten)

first fit (erste):

greift auf den 1. freien Speicher zu, egal wie groß der Block ist, sofern er größer als der geforderte Platzbedarf ist (z.B. bei 30 Byte 60 o. 100 Byte)

```

class stack:protostack
{
    friend ...
    private: int *top;
            int *bottom;
            static int a;
    public: stack::stack
            {
                top=new int[100];
                bottom=top;
            }
            stack::~stack
            :
            :
    virtual void stack::push(int x)
    stack s;
        
```

erbt von

ps □
↓
stack □

C von STROUSTRUP
C++ um OOP erhöht
(i++ um 1 erhöhen)

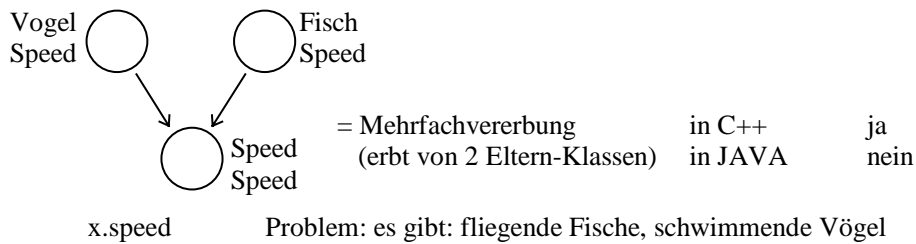
a gibt es nur 1x pro Klasse
→ Klassenvariable

vorsichtshalber schreibt man die Klasse vor die Methode

friend: Methode einer fremden Klasse oder die fremde Klasse selbst darf zugreifen

push wird dyn. gebunden, nicht stat. (JAVA immer dyn.)

Constructor wird automatisch aufgerufen



4.16 Programmierung in JAVA

bei JAVA:

- keine Mehrfachvererbung
- keine Freunde
- alles ist virtual (keine extra Eingabe von virtual)
- hat garbage collection (automatisch; destructors werden fast nicht benötigt)

Klasse in JAVA:

- Attribute
- Methoden
- Exception (Fehlerliste; catch = Stückchen Programm, das den Fehler bearbeitet)

```
class auto extents fahrzeug {
    private float speed;
    private boolean motor;
    exc bertl;
    public void beschl
    {
        speed=speed+10.0
    }
    public auto
    {
        speed=0.0
    }
}
...
catch bertl ...
```

Aufruf z.B.: if (x=0) raise bertl